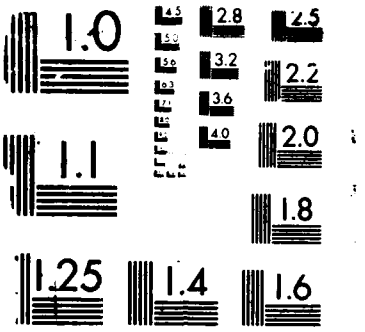
AD-A188 354 THE TEMPIS PROJECT: PERFORMANCE ANALYSIS OF TEMPORAL QUERIES(U) NORTH CAROLINA UNIV AT CHAPEL HILL DEPT OF COMPUTER SCIENCE I AHN ET AL 11 AUG 87 1/1

UNCLASSIFIED N00014-86-K-0680 F/G 12/5 NL



MICROCOPY RESOLUTION TEST CHART

4

Contract N00014-86-K-0680

The TEMPIS Project

Performance Analysis of Temporal Queries

Ilsoo Ahn[†] and *Richard Snodgrass*[‡]

August 11, 1987

Abstract

Temporal databases maintaining history data on line extend conventional databases with capabilities for *historical queries* and *rollback operations*. To analyse the performance of temporal queries on the database using various access methods, we propose a model consisting of four transformations through a series of intermediate expressions based on characteristics of database/relations and storage devices. We validate the model by comparing the I/O cost estimated from the analysis using the model with the actual cost measured from a prototype temporal DBMS. Since conventional databases are subsets of temporal databases, the model can also be used to analyze the performance of conventional databases.

TempIS Document No. 17

Department of Computer Science
University of North Carolina
Chapel Hill, NC 27514

This research was supported by NSF grant DCR-8402339.
‡ The work of this author was also supported by an IBM Faculty Development Award and by ONR contract N00014-K-0680.
† Present address of this author is AT&T Bell Laboratories, Columbus, OH 43213.

1. Introduction

Conventional databases store only the latest snapshot of the enterprise, and lack the capability to record and process time-varying aspects of the real world. Steady progress in disk storage technology, in terms of both capacity and cost, coupled with emerging new technologies such as optical disks has drawn attention to database management systems with temporal support or version management. Bibliographical surveys, however, show that most effort has focussed on conceptual aspects such as modeling, query languages, and the semantics of time [Bolour et al. 1982, McKenzie 1986]. Little has been written on issues concerning the implementation of temporal databases.

The performance of a temporal database management system (TDBMS) depends on many factors, including available access methods, the specific query processing strategy, and the size and composition of the stored data. There are two basic methods available to characterize the effectiveness of a TDBMS, the *empirical approach* and the *analytic approach*.

In the empirical approach, a system is implemented, and the actual performance on typical queries is instrumented. The primary advantage is that, if the measurement was done correctly, the performance measures can be trusted. The disadvantage is that the effort required for implementation is quite large, restricting implementation to at most a few access methods.

In the analytic approach, a mathematical characterization of the performance of a TDBMS is developed. Such a model can predict the performance of queries. This approach mirrors the empirical one in that the effort is much smaller, but the validity of the results are more questionable. We adopt the analytic approach, and propose a model to analyze the input and output cost of temporal queries on various access methods. The model consists of four transformations through a series of intermediate expressions based on the characteristics of database/relations and storage devices. We validate the model by comparing the I/O cost estimated from the analysis using the model with the actual cost measured from a prototype temporal DBMS.

In Section 2, we define the terms *temporal database* and *temporal query*, then review previous work on performance models. In Section 3, we present the details of our model, defining the intermediate expressions and describing the transformations. In Section 4, we show examples to analyze the I/O cost of temporal queries using the model, and also compare the cost estimated from the analysis with the actual cost measured from the prototype temporal database system described elsewhere [Ahn & Snodgrass 1986]. Finally in Section 5, we discuss how the model can be built into a system to automate performance analysis or optimization.

2. Previous Work

The term *temporal database* in the generic sense refers to databases with some degree of support for storing and processing time-dependent data [Ariav & Morgan 1982]. Examples include engineering databases containing a collection of design versions, personnel databases containing the history of employee records, and statistical databases containing time series data from scientific experiments.

If we look into the characteristics of time supported in these databases, we can identify three orthogonal kinds of time, *valid time*, *transaction time*, and *user-defined time* [Snodgrass & Ahn 1985, Snodgrass & Ahn 1986]. Depending on the capability to support either or both of valid time and transaction time, databases are classified into four types: *snapshot*, *rollback*, *historical*, and *temporal*. Snapshot databases are conventional databases without temporal support. Rollback databases support transaction time, recording the history of database activities. Historical databases support valid time, recording the history of a real world. Databases supporting both kinds of time are termed temporal databases in the narrower sense, as is used in this paper, to emphasize the importance of both kinds of time in database management systems.

In the past few years, approximately a dozen query languages have been designed to extract information from a time-varying databases. In this paper, we adopt *TQuel* [Snodgrass 1987] to express temporal constructs such as historical queries and rollback operations. TQuel extends Quel [Held et al. 1975] to provide query, data definition, and data manipulation capabilities supporting all four types of databases. It expresses historical queries by augmenting the **retrieve** statement with the **when** predicate to specify temporal relationships among participating tuples, and the **valid** clause to specify how the implicit time attributes are computed for result tuples. The rollback operation is specified by the **as of** clause for the rollback or the temporal databases. These added constructs utilize temporal relationships such as **precede**, **overlap**, **extend**, **begin of**, and **end of**.

TQuel augments the **append**, **delete**, and **replace** statements with the **valid** and the **when** clauses in a similar manner. It also supports the **create** statement to specify the temporal type of a relation. A formal tuple calculus semantics has been developed for all the TQuel statements. Since TQuel is a superset of Quel, both syntactically and semantically, all legal Quel statements are also valid TQuel statements.

A series of models for storage structures, with varying complexity and descriptive power, have been proposed over the past 15 years. In the remainder of this section, we review the major models to provide the motivation for our proposed model. Hsiao and Harary proposed one of the first formal models to analyze and evaluate generalized file organizations [Hsiao & Harary 1970]. The model represents the directory of a file with a set of sequences $(K_i, n_i, h_i; a_{i1}, a_{i2}, ..., a_{ih_i})$ for each keyword K_i, where n_i is the number of records containing the keyword, h_i is the number of sublists holding such records, and a_{ih_i} is the starting address of the h_i'th sublist. By varying the number and the length of sublists for each keyword, it can represent structures such as multilist files, inverted files, indexed sequential files, and some combinations of the files.

Severance, noting that this one dimensional model is unable to represent files which are not strictly list oriented, introduced a *two-dimensional* model [Severance 1975]. One dimension is whether the successor node is physically contiguous (*address sequential*) or connected through a pointer (*pointer sequential*). The other dimension is whether there is an index for the data (*data indirect*) or not (*data direct*). The four corners of this two-dimensional space represent sequential files, inverted files, list files, and pointer sequential inverted files.

Yao observed that Severance's model represents only a one-level index, imprecisely models indexed sequential files, and cannot model cellular list organizations [Yao 1977]. Instead, Yao represented the process of searching a file by an access tree composed of hierarchical levels comprised of attributes, keywords, accession lists, and virtual records. Based on this access path model, generalized access algorithms and cost functions for search and retrieval were presented. He also presented a file retrieval algorithm and an associated cost function for a single file query in a disjunctive normal form. Some of the parameters for the query were the total number of attributes and the average number of conjuncts in a query. Since this model has the underlying structure of the tree shaped access path, it is suitable for directory based file organizations such as inverted files, but is less applicable to files with other structures.

Yao later proposed a model representing the systematic synthesis of a large collection of access strategies for two relation queries [Yao 1979]. He identified 11 basic access operators such as restriction, join, record access, and projection, then presented without derivations cost equations for each operator measured in terms of page accesses. Permuting these operators gave 7 classes of processing algorithms for each relation, and 339 different algorithms for two relation queries, whose cost could be computed from cost equations of each operator. He modeled the storage structures with parameters indicating the existence of clustering, parent, child or chain links among relations, and the existence of clustering or non-clustering index for each attribute of relations. However, the model of [Yao 1977] was not used for this study.

Batory and Gotlieb proposed a *unifying* model, which decomposes physical databases into *simple files* and *linksets* [Batory & Gotlieb 1982]. The model for simple files characterizes the structures of records in a single file with a set of parameters such as design parameters, file parameters, and cost parameters. The model for linksets describes relationships between records in two simple files with parameters such as parent, child, cell size, and implementation methods. Basic operations and associated cost functions were also defined for simple files and linksets. This model is based on a collection of parameters that can describe the results of analyzing individual file organizations. Batory later augmented the unifying model with a *transformation model* that aids the process of conceptual-to-internal mappings [Batory 1985]. The model defines a set of *elementary transformations* that decompose conceptual files and links to their underlying internal files and linksets.

The models have made significant contributions in evaluating the performance of file organizations and access methods. Most of these models, however, were concerned with file structures; none of them has addressed the whole problem of evaluating the access cost given relational, let alone temporal, queries as input. Furthermore, special characteristics of query processing and access methods for temporal databases were beyond the scope of the existing models.

3. A New Model

Performance analysis of a database management system involves many factors such as query processing strategy, characteristics of stored data, access methods, and storage devices. We want to analyze the input and output cost for temporal queries on the database using various access methods. Hence we need a model which can characterize various phases of query processing in temporal database management systems. For this purpose, we

developed a model consisting of four transformations through a series of intermediate expressions based on the characteristics of database/relations and storage devices. We describe the details of the intermediate expressions and the transformations.

3.1. Transformation to the Algebraic Expression

TQuel is a language based on the tuple calculus, that is non-procedural. To describe the process of evaluating TQuel queries procedurally, we first define the *algebraic expression*. Then we discuss how a temporal query is transformed to an algebraic expression.

3.1.1. Algebraic Expression

An *algebraic expression* (AE) consists of *algebraic operators* and *connectives*. Algebraic operators are of three types: *conventional*, *temporal*, and *auxiliary*. These operators are more abstract than other temporal algebras [Clifford & Tansel 1985, McKenzie 1988, Navathe & Ahmed 1986]. The reason is that, while the other algebras are based on a single representation (*e.g.* tuple time stamping or attribute time stamping), we want our operators to accommodate many alternative representations.

The conventional relational operators include **`Select`**, **`Project`**, **`Join`**, **`Union`** and **`Difference`**. These operators perform the same actions as their snapshot counterparts; any temporal information is handled identically to the values from the non-temporal attributes. **`Select`** has two parameters: a relation and a predicate to specify the constraint that result tuples must satisfy. **`Project`** takes as parameters a relation and a set of attributes to be extracted from the relation. **`Join`** is to perform θ-*join* of two relations given as the first two parameters. The third parameter, the *join method*, specifies how to perform the *join* operation, since there are many ways to perform the operation. The fourth parameter is the predicate specifying how to combine information from two relations. Both **`Union`** and **`Difference`** take two relations as parameters, performing set addition and set subtraction respectively.

Temporal operators are included for temporal query constructs in TQuel. **`When`** performs *temporal selection* on a relation according to a temporal predicate applied to the values of *valid time* attributes. **`AsOf`** also performs temporal selection on a relation, but takes two time constants as parameters to compare with the values of *transaction time* attributes. **`Valid`** performs *temporal projection*, determining the value of the attribute *valid from*, *valid to*, or *valid at*.

Auxiliary operators are introduced to account for miscellaneous operations that do not change the query result but affect the query cost significantly. **`Temporary`** is used to create and access a temporary relation for the result of the operation marked by its parameter. **`Sort`** is used to sort tuples in the relation specified by the first parameter, using the remaining parameters as the key attributes for sorting. **`Reformat`** is used to change the structure of the relation specified by the first parameter to the form given by the second parameter, using the remaining parameters as the key attributes.

These algebraic operators can be combined together through *connectives* which specify information on ordering and grouping of the component operators. Two operators may be ordered in sequence, expressed as

$$\langle expression_1 \rangle \ ; \ \langle expression_2 \rangle$$

specifying that $\langle expression_1 \rangle$ should complete execution before $\langle expression_2 \rangle$ starts. Or they may be in parallel, denoted by

$$\langle expression_1 \rangle \ , \ \langle expression_2 \rangle$$

when two evaluations can proceed concurrently. Grouping of operators to delimit a query is denoted by a pair of braces, '{' and '}', while a pair of square brackets, '[' and ']', represent a set of operators which can be evaluated simultaneously for each tuple. These connectives can characterize different strategies for evaluating a query expressed by a combination of algebraic operators.

An operator may have a *label* which can be referred to in other operators such as **`Temporary`**. By using labels, we can eliminate deeply nested parentheses common in algebraic descriptions of a query. Thus an algebraic expression, describing TQuel queries in a procedural form, is a combination of labels, algebraic operators with appropriate parameters, and connectives. Abbreviated BNF syntax of the algebraic expression is shown in Appendix A.1.

3.1.2. Transformation

A query in TQuel can be expressed in terms of algebraic expressions. In general, there are several of such mappings that provide the same result, but exhibit different I/O costs. For example, the query

```
range of    h  is relation_h
retrieve    (h.id, h.seq)      where h.id = 500
```

can be mapped to

```
AE-1:       { L1:   Select     (h, h.id = 500);
                    Project    (L1, h.id, h.seq) }
```

This expression selects tuples with `id = 500` from the relation `h`, then extracts attributes `id` and `seq` from the result of the previous operation labeled as `L1`. Since the two operations are separated by ';', they must execute sequentially. The same query can be mapped to

```
AE-2:       {[ L1:  Select     (h, h.id = 500);
                    Project    (L1, h.id, h.seq) ]}
```

This expression is similar to AE-1, but specifies that **Select** and **Project** can be evaluated together (still sequentially) for each tuple. Thus the need for a temporary file to store intermediate results between the two operations is explicitly eliminated.

For the query

```
range of    h  is relation_h
range of    i  is relation_i
retrieve (h.id, i.id, i.amount)
     where h.id = i.amount
      when  h overlap i and   i overlap "now"
```

we can list three different algebraic expressions.

```
AE-3:  {  L1:  Join       (h, i, TS, h.id = i.amount and h overlap i);
          L2:  When       (L1, i overlap "now");
               Project    (L2, h.id, i.id, i.amount)     }
```

This expression specifies **Join** using *tuple substitution* (TS) of two relations, `h` and `i`, followed by temporal selection **When**, followed by **Project**, all in sequence. Another example:

```
AE-4:  {  L1:  When       (i, i overlap "now");
          L2:  Project    (L1, i.id, i.amount, i.valid_from, i.valid_to);
          L3:  Join       (h, L2, TS, h.id = i.amount and h overlap i);
               Project    (L3, h.id, i.id, i.amount)     }
```

This expression is functionally equivalent to AE-3, but differs, perhaps markedly, in performance.. AE-4 specifies that the **When** operation is first executed to select tuples from the relation `i` whose `valid to` attribute is `"now"`, then four attributes are extracted from the result tuples, then the result is joined with the relation `h`, and finally three attributes are extracted. However, AE-4 does not provide information on what operations can proceed together and whether a temporary relation is needed.

```
AE-5:  {[ L1:  When       (i, i overlap "now");
          L2:  Project    (L1, i.id, i.amount, i.valid_from, i.valid_to)];
          L3:  Temporary  (L2);
        [ L4:  Join       (h, L3, TS, h.id = i.amount and h overlap i);
               Project    (L4, h.id, i.id, i.amount)     ]}
```

This expression is similar to the previous expression AE-4, but specifies that **When** and **Project** can be evaluated together on each tuple, the intermediate result is stored into a temporary relation, and **Join** and **Project** can also be performed together.

3.2. Transformation to the File Primitive Expression

In this section, we define the *file primitive expression* to characterize the input and output activities involved in processing a query. We also discuss how to represent information on the characteristics of data stored in database/relations. We then describe how the algebraic expression is transformed to the file primitive expression.

3.2.1. File Primitive Expression

A file primitive expression represents the process of accessing a file in terms of two file primitives: **Read** and **Write**. Both of the primitives take parameters such as the access method, the size of a file, or the length of the overflow chain. The access methods include `heap`, `hash`, `isam` and `btree`. Such an expression provides more detail than an algebraic expression how a query is evaluated.

Primitives are combined to form an arithmetic expression, called the file primitive expression (FPE), to describe the situation when one or more primitives are repeated or executed together to perform an algebraic operation. The primitives take an access method and one or more constants denoting, for example, the size of a file or the length of the overflow chain. The syntax of the file primitive expression is shown in Appendix A.2.

For example, a file primitive expression may be as simple as:

```
FPE-1:   Read  (Hash, 0)
```

specifying one hashed access without any overflow records, or more complex like

```
FPE-2:    Read  (Heap, 128)              +
        ( Read  (Heap, 19) * 2 - 1       +
          Write (Heap, 19) * 3 - 1 )     +
          Read  (Heap, 19)               +
          Read  (Hash, 0) * 1024
```

specifying one *read* from the heap of 128 blocks, two *read*'s from the heap of 19 blocks, three *write*'s to the heap of 19 blocks, another *read* from the heap of 19 blocks, and finally a hashed access repeated 1024 times.

3.2.2. Characteristics of Database/Relations

To transform the algebraic expression to the file primitive expression, we need information on the characteristics of relations comprising a database. Typical catalog relations in conventional DBMS's hold information for all relations such as relation names, temporal types, storage structures, attribute counts, attribute names, attribute formats, attribute lengths, key attributes, tuple lengths, and tuple counts.

Additional information on the data contents is needed so that the algebraic expression can be transformed to the file primitive expression. Examples are selectivity and distribution of attribute values, volatility of data, and the update count in case of a temporal database. Various parameters for each relation and attribute are specified. BNF syntax to represent the characteristics of database/relations is given in Appendix A.3.

It is a difficult problem to estimate the response set of a query and the number of block accesses without actually examining stored data, though there has been significant research on the subject. Inaccurate characterization of stored data may account for a large portion of the discrepancy, if any, between the analysis result and the actual measurement.

3.2.3. Transformation

We can transform the algebraic expression to the file primitive expression. For example, the algebraic expression AE-2 can be transformed to the file primitive expression FPE-1 shown earlier, assuming that the relation `h` is hashed on the attribute `id` with no overflow records.

There are a large number of valid combinations for algebraic expressions even for conventional snapshot databases. The problem becomes more complicated when historical queries and rollback operations for temporal databases are introduced. Rather than listing all the possible algebraic expressions, we identify basic constructs occurring in temporal queries, and transform the subset of algebraic expressions, composed of such constructs, to file primitive expressions. The transformation is dependent on the characteristics of database/relations described above.

Algebraic operators involve either one relation or two relations. **Select**, **Project**, **When**, **AsOf**, **Valid**, **Temporary**, **Sort**, and **Reformat** operate on one relation, while **Join**, **Union**, and **Difference** operate on two relations. The characteristics of each operator is discussed in terms of the file primitive expression.

- **Select** (*relation, predicate*)
  The first parameter *relation* is the base relation for the operation, and the second parameter *predicate* specifies constraints on the *relation* that result tuples must satisfy. Performance of **Select** depends on various factors such as the structure of the *relation*, the type of the *predicate*, and the characteristics of data stored in the *relation*.

  (1) If the *predicate* fully specifies a key for a random access path existing for the *relation*, the file primitive expression is:

  Read (*access path, n*)

  where the *access path* may be `hash`, `isam`, `btree` or other desired access method. The second parameter n is the length of the overflow chain, which is determined from the characteristics of database/relations.

  (2) Otherwise, the file primitive expression is:

  Read (Heap, b)

  where b is the size of the *relation* in blocks, meaning the relation is sequentially scanned.

- **Project** (*relation, attribute list*)
  This operation scans the *relation* to extract a list of attributes, *attribute list*, hence its file primitive expression is:

  Read (Heap, b)

  where b is the size of the *relation* in blocks.

- **Join** (*relation*$_1$, *relation*$_2$, *join method, predicate*)
  There are several methods to perform a join, such as TS (*tuple substitution*), BS (*block substitution*), and SM (*sort & merge*). Let
  t_1 : the number of tuples in *relation*$_1$
  t_2 : the number of tuples in *relation*$_2$
  b_1 : the size of *relation*$_1$ in blocks
  b_2 : the size of *relation*$_2$ in blocks
  Each method is briefly described with the corresponding file primitive expression.

  (1) TS : tuple substitution method
  Each tuple in the smaller relation is substituted to select tuples from the other relation satisfying the *predicate*.

  Read (Heap, b_1) +
  $FPE_2 * t_1$

  assuming $t_1 < t_2$. FPE_2 is the file primitive expression for

  Select (*relation*$_2$, *predicate'*)

  where *predicate'* is the predicate with the tuple variable for *relation*$_1$ replaced by each tuple in *relation*$_1$.

  (2) BS : block substitution method
  For each block in the smaller relation, the other relation is scanned. In this process, all tuples in one block of each relation are joined according to the *predicate*. It is faster than *tuple substitution* especially when there is no random access path to evaluate the *predicate*.

  Read (Heap, b_1) +
  Read (Heap, b_2) * b_1

  where $b_1 < b_2$.

(3) SM : sort & merge method

Each relation is sorted, then the resulting relations are scanned in parallel to merge tuples satisfying the *predicate*.

```
Read  (Heap, b1)                          +
Read  (Heap, b2)                          +
FPE (Sort (relation1, attribute list) )   +
FPE (Sort (relation2, attribute list))
```

where *FPE* (**Sort** (...)) is the file primitive expression for **Sort** to be described later, and *attribute list* is the list of attributes participating in the *predicate*. If both relations are already in order, the file primitive expression is simply

```
Read  (Heap, b1) +
Read  (Heap, b2)
```

- **Union** (*relation*$_1$, *relation*$_2$) and **Difference** (*relation*$_1$, *relation*$_2$)

  Both operators need to scan two relations, so the file primitive expression is

```
Read  (Heap, b1) +
Read  (Heap, b2)
```

  where b_1 and b_2 are the sizes of *relation*$_1$ and *relation*$_2$, respectively, in blocks.

- **When** (*relation, temporal pred*)

  When is similar to **Select**, where the temporal predicate, *temporal pred*, is restricted to a single variable predicate specifying the constraint on the *valid time* attributes that result tuples must satisfy. Hence the file primitive expression is, like **Select**,

```
Read  (access path, b)
```

  or

```
Read  (Heap, b)
```

  depending on the type of the *predicate*, and the existence of a random access path to satisfy the temporal predicate.

- **AsOf** (*relation*, t_1, t_2)

  AsOf is identical to

```
Select (relation, t1 ≤ transaction_stop  and  transaction_start ≤ t2)
```

  Hence the file primitive expression is similar to that for **Select**.

- **Valid** (*relation, From/To/At, temporal expr*)

  Valid is similar to **Project**, where the temporal expression, *temporal expr*, is restricted to a single variable expression with the domain of time values. The file primitive expression is

```
Read  (Heap, b)
```

  where *b* is the size of the *relation* in blocks.

- **Temporary** (*label*)

  This operator, as shown in AE-5, is to create a temporary relation, and to store the intermediate result from the previous operation marked by the *label*. Its file primitive expression is in general:

```
( Read  (Heap, b) * kr - lr       +
  Write (Heap, b) * kw - lw  )
```

  where *b* is the number of blocks in the resulting relation, and k_r, l_r, k_w, l_w are implementation dependent constants. In a prototype used for comparison, each block, except the last one, of a temporary relation is read twice and written three times, so $k_r = 2$, $k_w = 3$, and $l_r = l_w = 1$.

```
( Read  (Heap, b) * 2 - 1      +
  Write (Heap, b) * 3 - 1 )
```

- **Sort** (*relation, attribute list*)
  This is used to sort the *relation* using a list of attributes, *attribute list*, as key attributes for sorting. Since it takes $O\ (b \times \log_m b)$ block accesses to sort a file of b blocks using the m-way sort-merge, the file primitive expression is in general:

$$\begin{aligned} &\textbf{Read}\ \ (\text{Heap},\ b_1)\ *\ O\ (\log_m b_1)\ + \\ &\textbf{Write}\ (\text{Head},\ b_1)\ *\ O\ (\log_m b_2)\ + \\ &\textbf{Read}\ \ (\text{Heap},\ b_2)\ *\ O\ (\log_m b_2)\ + \\ &\textbf{Write}\ (\text{Head},\ b_2)\ *\ O\ (\log_m b_2) \end{aligned}$$

- **Reformat** (*relation, storage spec, attribute list*)
  This is to reformat the *relation* to the storage structure, *storage spec*, using a list of attributes, *attribute list*, as key attributes. Its file primitive expression is in general:

$$\begin{aligned} (\ &\textbf{Read}\ \ (\text{Heap},\ b) \quad\quad\quad + \\ &\textbf{Write}\ (\text{Heap},\ b) \quad\quad\quad + \\ &\textit{FPE}\ \ (\textbf{Sort}\ \ (\textit{relation, attribute list})\ ) \end{aligned}$$

  where *FPE* (**Sort** (...)) is the file primitive expression for **Sort** in case we need to sort the *relation* for reformatting.

Thus far, each operator has been discussed in terms of file primitive expressions. An algebraic expression with multiple operators can be transformed to the file primitive expression which is the sum of the file primitive expressions for the component operators. An exception to this rule is the case when **Project** or **Valid** follows **Select**, **Join**, or **When**, and the two operations are grouped together by a pair of square brackets. In this case, the file primitive expression is simply that of the first operation. For example, an algebraic expression

```
[[ L1:  Select    (h, id = 500);
        Project   (L1, h.id, h.seq) ]]
```

is transformed to

```
Read  (Hash, 0)
```

performing **Project** effectively for free.

In summary, an algebraic expression is transformed to a file primitive expression, which contain **Read** and **Write** operations, by replacing each algebraic operator according to the rules shown above. The resulting expression may contain constants, whose values are supplied by the characteristics of database/relations.

3.3. Transformation to the Access Path Expression

We define the *access path expression* which represents the path taken through the storage structure to satisfy an access request represented by a file primitive expression. An access path is usually confined to a single file, but it may involve more than one file, which is the case with storage structures for temporal databases discussed elsewhere [Ahn 1986]. We section first describe the access path expression for a single file, and then extend it for multiple files.

3.3.1. Access Path Expression for a Single File

The conceptual unit of an access in the access path expression is a *node*, which consists of one or more *physically contiguous* records participating in the access. The node itself consists of one or more *records*, depending on the underlying storage structure.

A set of nodes are connected together to make up an access path either *directly* or *indirectly*. In simple cases, an access path is directly represented as a set of nodes. In other cases, it helps to conceptualize an access path as being composed of some components, each of which is itself a set of nodes. This process of *hierarchical decomposition* may proceed for as many levels as useful.

The process of decomposition is restricted to three levels, which is sufficient to describe the storage structures discussed in this paper. However, it is straightforward to extend it to incorporate more levels. In this three level hierarchy, a set of nodes are grouped to make up a *chain*, and a set of chains compose an access path. Therefore, an access path through a single file, or simply a *file path*, is represented as a set of chains, each of which is a set of nodes. As mentioned above, each node itself consists of one or more records.

The access path expression identifies a fixed number of *modes*, specifying how components such as nodes, chains, or file paths are connected with one another. We can classify the modes as either *guided* or *searched*.

Guided : If a random access mechanism exists to locate the component

- **H** : the address is computed by a *hash* function
- **P** : the address is provided by a *pointer*
- **A** : the component is physically *adjacent* to its predecessor
- **S** : the component shares the *same* starting address with its higher level component
- **M** : the component is in the *main memory*

Searched : If no random access mechanism exists

- **O** : the file is *ordered*, so logarithmic search is possible
- **U** : the file is *unordered*, so sequential search is necessary.

This process of hierarchical decomposition, decomposing an *access path* or a *file path* into *chains*, a chain into *nodes*, and a node into *records*, is captured into a single expression called the *access path expression (APE)*. A canonical form for an access path expression, whose syntax is shown in Appendix A.4, is

$$(Mode \quad count_1 \ (Mode \quad count_2 \ (Mode \quad count_3)^+ \ )^+ \ )$$

where

$count_1$ is the number of chains in the file path,
$count_2$ is the number of nodes in the chain, and
$count_3$ is the number of records in the node.

As described earlier, the components in the three level hierarchy are the access path, chains, and nodes. Each component is described by a (*Mode*, *count*) pair, where the *mode* tells how to locate the component, and the *count* shows the number of subcomponents in it. Then the (*Mode*, *count*) pair is followed by a list of descriptors for its subcomponents enclosed in parentheses. The level of a component in the hierarchy is determined by the depth of enclosing parentheses. The outermost parentheses represent the access path, while the innermost parentheses represent a node which is defined to consist of records.

Each subcomponent is described in sequence, but if all the successors of a certain subcomponent are the same, they need not be repeated. Therefore, if the number of descriptors is smaller than the specified count, the remaining subcomponents are assumed to have the same descriptor as the last one. When a component has only one subcomponent and the mode of the subcomponent is **S** (meaning the subcomponent shares the same starting location), the extra level of decomposition does not provide any further information, and may be omited.

In the access path expression, a set of file parameters are used to quantify physical properties of a file. Some of the parameters are:

- f : number of records in a file
- b : number of records in a block
- r : number of bytes in a record, and
- n : number of records to be accessed.

Some examples of access path expressions are described now for various access methods.

Example-1. Scanning a sequential file:

The access path can be considered as an unordered collection of f records. The access path expression is:

$$(\mathbf{U} \ f)$$

Since the head of the path expression is **U**, the path needs to be searched sequentially. The access path can also be regarded as consisting of a single node, which has f records. Then the expression becomes:

$$(\mathbf{U} \ 1 \ (\mathbf{S} \ f))$$

We can follow the three level hierarchy by introducing the level of *chain*. Then the access path has a single chain, which has one node. The node itself consists of f records.

$$(\mathbf{U} \ 1 \ (\mathbf{S} \ 1 \ (\mathbf{S} \ f)))$$

Example-2. Accessing a hashed file without an overflow:

$$(\mathbf{H} \ 1) \equiv (\mathbf{H} \ 1 \ (\mathbf{S} \ 1)) \equiv (\mathbf{H} \ 1 \ (\mathbf{S} \ 1 \ (\mathbf{S} \ 1)))$$

This is similar to Example-1 except that the head of the access path is located through hashing, and that a

node is of one record.

Example-3. Accessing an inverted file as shown in Figure 1 (a):

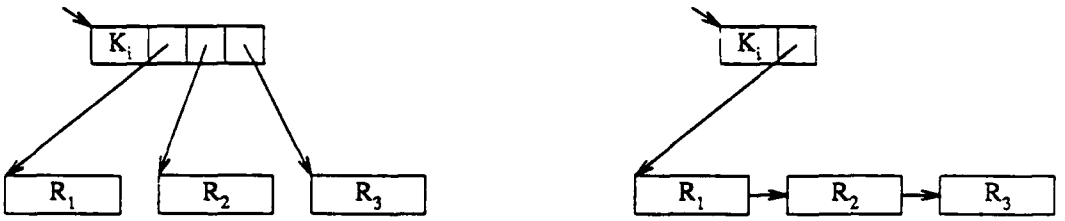
$$(\mathbf{P}\ 3\ (\mathbf{P}\ 1\ (\mathbf{S}\ 1))\ (\mathbf{P}\ 1\ (\mathbf{S}\ 1))\ (\mathbf{P}\ 1\ (\mathbf{S}\ 1)))$$

The path, whose head is located through a pointer, contains a key value and three chains. Each chain is also located through a pointer, and each has one node. Each node shares the same address with the chain, and is of one record. Since all the chains are identical, we need not repeat the descriptor for each chain. Then the expression is abbreviated to:

$$(\mathbf{P}\ 3\ (\mathbf{P}\ 1\ (\mathbf{S}\ 1)))$$

In general, there will be n chains:

$$(\mathbf{P}\ n\ (\mathbf{P}\ 1\ (\mathbf{S}\ 1)))$$



(a) an Inverted File (b) a Multilist File

Figure 1: Structures for an Inverted File and a Multilist File ($n = 3$)

Example-4. Accessing a multilist file, as shown in Figure 1 (b):

$$(\mathbf{P}\ 1\ (\mathbf{P}\ 3\ (\mathbf{S}\ 1)\ (\mathbf{P}\ 1)\ (\mathbf{P}\ 1)))$$

The path, whose head is located through a pointer, has one chain. The chain is located through a pointer, and has three nodes, each of which has one record. The first node shares the same address as the chain, and the subsequent nodes are located through pointers. Since the second node and the third node are identical, the expression can be abbreviated to:

$$(\mathbf{P}\ 1\ (\mathbf{P}\ 3\ (\mathbf{S}\ 1)\ (\mathbf{P}\ 1)))$$

In general, there will be a chain of n nodes:

$$(\mathbf{P}\ 1\ (\mathbf{P}\ n\ (\mathbf{S}\ 1)\ (\mathbf{P}\ 1)))$$

Note the difference from the expression for an inverted file in Example-3.

Example-5. Accessing a cellular inverted file, where each node is a cellular block of size b:

$$(\mathbf{P}\ \frac{n}{b}\ (\mathbf{P}\ 1\ (\mathbf{S}\ b)))$$

This example is similar to Example-3, but the path has $\frac{n}{b}$ chains. Each chain has one node, which consists of b records.

Example-6. Accessing a cellular multilist file, where each node is a cellular block of size b:

$$(\mathbf{P}\ 1\ (\mathbf{P}\ \frac{n}{b}\ (\mathbf{S}\ b)\ (\mathbf{P}\ b)))$$

Similar to Example-5, but the chain has $\frac{n}{b}$ nodes, each of which consists of b records. Note that we can

repeat the descriptor for the second node, (P b), $\frac{n}{b} - 1$ times.

Example-7. Accessing an ISAM file with the master index in core:

(M 1 (P 1 (P 1)))

An entry in the master index, which resides in the main memory, points to the head of a single chain, corresponding to a directory entry. The chain consists of a node, which consists of a single record. The head of the node is located through a pointer. If the file has an overflow chain of n nodes, each of which is a single record, the access path expression is:

(M 1 (P n+1 (P 1)))

Example-8. Accessing a hashed file with an overflow chain of n records:

(H 1 (P n (S 1) (P 1)))

The access path is located by hashing, and has a single chain. The chain has n nodes, each of which has a single record. The head of the chain is located through a pointer, and shares the same address with the head of the first node.

3.3.2. Access Path Expression for Multiple Files

Thus far, we have discussed access paths involving only one file. When two or more files are involved in an access, the *composite* access path is represented by the combination of the individual file paths. There are two criteria to determine the relationship between two files. One is *ordering*, which determines whether two files are ordered or not. If *ordered*, they are accessed in *serial*, where one file path always precedes the other one. If *unordered*, there is no restriction on ordering, so two files may be accessed in *parallel*. The other criterion is whether only *one* file needs to be accessed, or *both* files should be accessed. Obviously, if both files should be accessed, the *ordering* information between the two files must be known. With this restriction, the two criteria lead to five possible combinations as follows.

(1) [$FilePath_1$; $FilePath_2$]
Two files are accessed in serial, like the temporally partitioned storage structure discussed elsewhere [Ahn 1986].

(2) [$FilePath_1$, $FilePath_2$]
Both files need to be accessed, but there is no fixed ordering, like a horizontally partitioned relation [March & Severance 1977].

(3) [$FilePath_1$?; $FilePath_2$]
The first file is accessed. If it is unsuccessful, then the second file is accessed. An example is a differential file [Severance 1976].

(4) [$FilePath_1$?, $FilePath_2$]
Either of the two files is accessed. If it is unsuccessful, then the other file is accessed. An example is a vertically partitioned relation [Ceri & Pelagatti 1984].

(5) [$FilePath_1$? $FilePath_2$]
Only one of the two files needs to be accessed, and which one to access is known. An example is a differential file with the Bloom filter in main memory [Gremillion 1982]:

[(M 1) ; [$FilePath_1$? $FilePath_2$]]

Example-9. Accessing a file with *reverse chaining*:
An example of the temporally partitioned storage structure is *reverse chaining*, in which all history versions of each version set are linked in reverse order starting from the current version [Ahn 1986]. Once the current version is located in the current store, its predecessors can be retrieved without scanning the whole history store. Figure 2 shows the structure diagram.

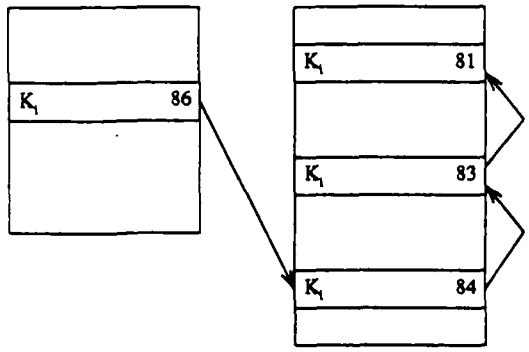


Figure 2: Reverse Chaining

The access path expression for this structure is:

$$[\ FilePath_1\ ;\ (\mathtt{P}\ n\ (\mathtt{S}\ 1)\ (\mathtt{P}\ 1))\ ]$$

where $FilePath_1$ is for the current store, and n is the number of history versions. This expression shows that there is a single chain. The head of the chain is located through a pointer, and the chain has n nodes. Each of the node is of one record, and is connected to the predecessor by a pointer.

Example-10. Accessing a path composed of three files:

It is also possible to involve more than two files in various combinations. If they are accessed in sequence like a three level store, the access path expression is:

$$[\ [\ FilePath_1\ ;\ FilePath_2\ ]\ ;\ FilePath_3\ ]$$

If they are in the shape of a tree, as in Figure 3 (a), file 1 is accessed first, then the other two files are accessed in any order. The access path expression is:

$$[\ FilePath_1\ ;\ [\ FilePath_2\ ,\ FilePath_3\ ]\ ]$$

In Figure 3 (b), files 1 and 2 are accessed in any order, then then the third file is accessed. The access path expression is:

$$[\ [\ FilePath_1\ ,\ FilePath_2\ ]\ ;\ FilePath_3\ ]$$

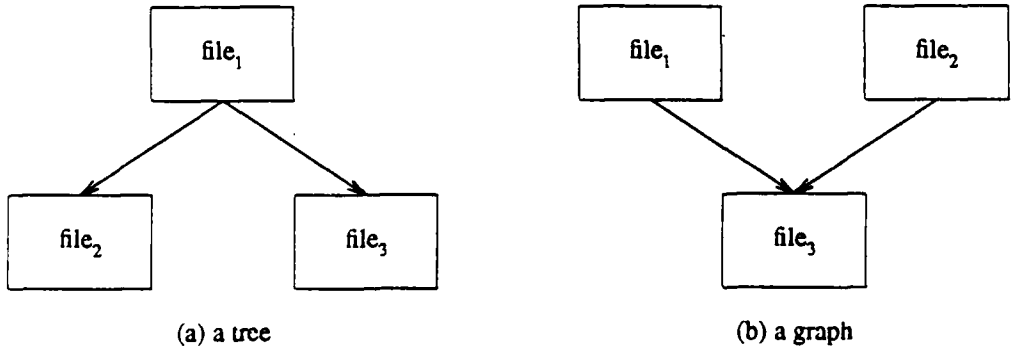


Figure 3: Access Paths with Three Files

BNF syntax of the access path expression involving multiple files is given in Appendix A.5.

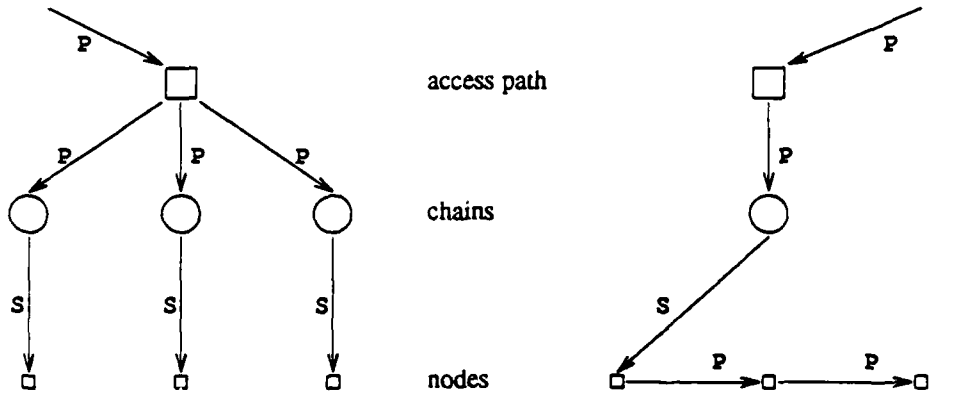
In summary, the access path expression represents access paths, taken through a storage structure to satisfy a request represented by a file primitive expression, with the access path expression augmented with a set of file parameters. The access path expression is simple and well-defined, yet versatile in representing a variety of access methods which may involve more than one file. The access path expression is loosely based on four models described in Section 2. It captures the concepts of the sublist in [Hsiao & Harary 1970], data direct/indirect and address/pointer sequential in [Severance 1975], hierarchy of levels in [Yao & Merten 1975] and a set of parameters in [Batory & Gotlieb 1982], but extends them significantly in a single framework.

3.4. Transformation to the Access Cost

From an access path expression, we can estimate the access cost in terms of random and sequential access counts. We can also calculate the elapsed time based on the characteristics of storage devices.

3.4.1. Access Count

Given an access path expression, it is possible, though not necessary, to parse the expression, and derive an *access path graph (APG)*. In the graph, each component is denoted as a vertex, while relationships among components are denoted as an edge marked with the associated mode. For an access path involving a single file, the graph results in a tree, with the vertex for *file path* as the root. Access path graphs for the access path expressions in Example-3 and Example-4 are shown in Figure 4. While there is a similarity between the structure diagram in Figure 1 and the access path graph in Figure 4, this is not always the case. For example, the structure diagram for a sequential file or a hashed file is not a graph, though the corresponding access path graph is. The access path graph is only conceptual, and not necessarily tied to the physical structure itself.



(a) an inverted file (Ex-3)

(P 3 (P 1 (S 1)))

(b) a multilist file (Ex-4)

(P 1 (P 3 (S 1) (P 1)))

Figure 4: Access Path Graphs ($n = 3$)

The access path graph not only visualizes the process of accessing files, but also represents the cost incurred in traversing an access path by the length of the path. In fact, it is possible to estimate the access cost in terms of random and sequential *access counts (AC)* either from the access path graph derived from the access path expression or directly from the access path expression, based on the modes to connect components. The rules to estimate the upper bound for the access count are:

H (Hashing)	: $(1 + \alpha)$ random accesses, where α is determined by the overflow handling method
P (Pointed)	: 1 random access
A (Adjacent)	: 1 sequential access
S (Same-as-before)	: no access cost
I (Main-memory)	: no access cost

O (Ordered)	: logarithmic search	$(O\ (log\ \frac{f}{b}))$
U (Unordered)	: sequential search	$(\frac{f+b}{2b})$

3.4.2. Characteristics of Storage Devices

There are many parameters affecting the performance of storage devices, such as the medium type, fixed or moving heads, read/write or write-once, seek time, transfer rate, number of cylinders-tracks-sectors, sector size, *etc.* For the purpose of this research, we adopt a representation to characterize the performance of storage devices with two parameters. They are t_{ra}, time needed to access a block randomly, and t_{sa}, time needed to access a block sequentially. Given the count of random and the sequential accesses, *e.g.* from the access path expression, it is possible to estimate the time required to satisfy the request.

For a typical moving head disk, the time needed to access a block randomly is the sum of seek time, rotational delay, and data transfer time.

$$t_{ra} = t_{seek} + t_{rd} + t_{tr}$$

The average seek time, t_{seek}, assuming uniform distribution of seek distances, is:

$$E\ (t_{seek}) = \sum_{i=1}^{c-1} t_i \times \frac{2(c-i)}{c^2 - c}$$

where t_i is the seek time for distance over i cylinders, and c is the total number of cylinders for the disk [Wiederhold 1981]. The average rotational delay, t_{rd}, is the time for one half revolution, and the data transfer time, t_{tr}, is the block size divided by the data transfer rate.

Ideally, accessing a block sequentially is free of any head movement and even the rotational delay.

$$t_{sa} = t_{tr}$$

However, a *logically* sequential block may not be *physically* adjacent under many operating systems, *e.g.* Unix, which may allocate a block to a file randomly from the pool of free pages [Stonebraker 1981]. Even when the block is physically adjacent, it is highly probable in a multi-process system that another process sharing the disk disrupts the sequentiality by moving the head to another sector or cylinder.

Another factor to be considered is the difference between the block size of the database management system and the page size of the operating system. Let b_{db} be the block size of the database management system, and let p_{os} be the page size of the operating system. If b_{db} is bigger than p_{os}, it takes extra disk accesses to retrieve one database block. In the opposite case, which is actually the case in the prototype used for comparison, some sequential blocks are already in the main memory with the effect of *read-ahead*. If we let $n = \frac{p_{os}}{b_{db}}$, the average t_{sa} in a multi-process environment will be:

$$t_{sa} = \frac{1}{n}\ (t_{seek} + t_{rd} + n \times t_{tr})$$

3.4.3. Elapsed Time

An experiment was run to measure the average t_{ra} and t_{sa} on a moving head disk connected to a Vax/780. Here, the file used for sequential access was in fact physically contiguous. The result is shown in Figure 5. We use the time for an average load in estimating elapsed time to process sample queries in Section 4. Given the random and the sequential access counts, we can calculate the elapsed time using appropriate numbers from the table in Figure 5.

	Low Load	Average Load	High Load
Sequential	16.9	18.4	19.9
Random	24.8	31.3	37.8

Figure 5: Time (in *msec*) to Access a Block

4. Performance Analysis

With the model consisting of a series of transformations just described, it is possible to analyze the input and output cost to process TQuel queries. Any complex query involving more than two relations can be decomposed into simpler queries of two or less relations [Wong & Youssefi 1976]. Hence a TQuel query can be transformed to an algebraic expression, which consists of algebraic operators involving one or two relations, representing the strategy used to process the query. This step is performed by the parser and the semantic analysis portion of a TQuel query processor. While our model abstracts the details of various representational decisions, it does not encode query processing strategies such as decomposition nor query optimization techniques such as moving a selection across a cartesian product. These strategies are encapsulated in the query processor.

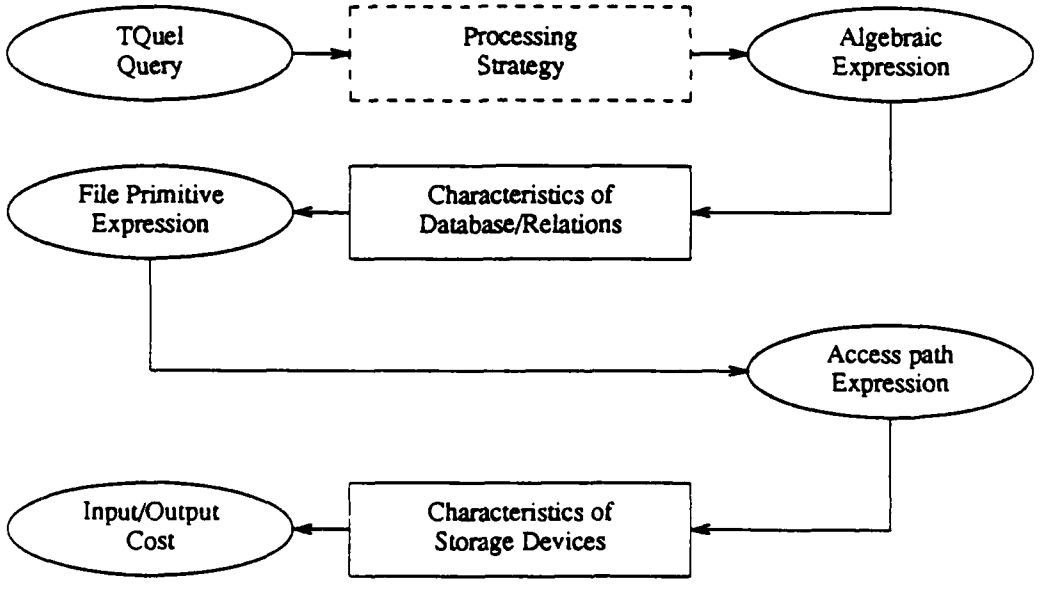


Figure 6: Performance Analysis with the Model

The algebraic expression is then transformed into the file primitive expression based on the characteristics of database/relations. Next, the file primitive expression is transformed into the access path expression, and eventually to the access cost in terms of random and sequential access counts. Finally, the access count is converted to the elapsed time according to the characteristics of storage devices. These steps are illustrated in the Figure 6, where we show the *processing strategy* in a dotted box to denote that the processing strategy is not a part of our model.

4.1. Examples

We now show how we can use our model to analyze the performance of temporal queries in TQuel. For example, both of the algebraic expressions AE-1 and AE-2, shown in Section 3.1.2, represent the TQuel query:

```
range of    h  is relation_h
retrieve    (h.id, h.seq)     where h.id = 500
```

Since AE-2 provides more information on how to process the query, let's evaluate the input and output cost for AE-2 using our model. We first try the case where the characteristics of database/relations shows that `relation_h` is a hashed file with no overflow records. Then transforming the algebraic expression to the file primitive expression:

```
Read  (Hash, 0)
```

which is the same as FPE-1 shown in Section 3.2.1. This is in turn transformed to the access path expression:

```
(H 1)
```

whose access cost is

$$AC = C\,(\text{APE}) = C\,((\mathbf{H}\ 1)) = 1\ \textit{random access} = 31.3\ \textit{msec}$$

where the average time to perform 1 random access is found to be about 31.3 *msec* according to the characteristics of storage devices.

If the characteristics of database/relations shows that `relation_h` is a hashed file with 14 overflow records, then its file primitive expression becomes:

```
Read  (Hash, 14)
```

Now the corresponding access path expression is:

```
(H 1 (P 14 (S 1) (P 1)))
```

Its access cost is

$$AC = C\,(\text{APE}) = C\,((\mathbf{H}\ 1\ (\mathbf{P}\ 14\ (\mathbf{S}\ 1)\ (\mathbf{P}\ 1)))) = 15\ \textit{random accesses} = 470\ \textit{msec}$$

For another example, algebraic expressions AE-3, AE-4, and AE-5 can all be considered as representations of the TQuel query:

```
range of    h  is relation_h
range of    i  is relation_i
retrieve (h.id, i.id, i.amount)
      where h.id = i.amount
      when  h overlap i and   i overlap "now"
```

Let's choose AE-5, as shown in Section 3.1.2, and evaluate its input and output cost.

```
AE-5:  {[ L1:  When       (i, i overlap "now");
          L2:  Project    (L1, i.id, i.amount, i.valid_from, i.valid_to)];
          L3:  Temporary  (L2);
       [  L4:  Join       (h, L3, TS, h.id = i.amount and h overlap i);
               Project    (L4, h.id, i.id, i.amount)     ]}
```

First, assume that the characteristics of database/relations shows that `relation_h` is a hashed file and `relation_i` is an ISAM file, each without any overflow records. It also shows that the size of `relation_i` is 128 blocks, the size of the temporary relation is 19 blocks, and there are 1024 tuples in the temporary relation. Then AE-5 is transformed to to the file primitive expression:

```
      Read   (Heap, 128)             +
    ( Read   (Heap, 19) * 2 - 1      +
      Write  (Heap, 19) * 3 - 1 )    +
      Read   (Heap, 19)              +
      Read   (Hash, 0) * 1024)
```

which is in fact the same as FPE-2 shown earlier. The first **Read** primitive accounts for the **When** and the **Project** operations, the second **Read** and the **Write** primitives account for the **Temporary** operation, and the third and the fourth **Read** primitives account for the **Join** and the **Project** operations.

Now the **Read** operations in the file primitive expression are transformed to the access path expression for input:

```
(U 128)                +
(U 19) * 2 - 1         +
(U 19)                 +
(H 1) * 1024
```

Likewise, the **Write** operation in the file primitive expression is transformed to the access path expression for output:

```
(U 19) * 3 - 1
```

Now, the access cost for input is:

$$AC_i = C((\mathbf{U}\ 128)) + C((\mathbf{U}\ 19) * 2 - 1) + C((\mathbf{U}\ 19)) + C((\mathbf{H}\ 1)) * 1024$$
$$= 1028\ \textit{random accesses} + 180\ \textit{sequential accesses} = 35.5\ \textit{sec}$$

and the access cost for output is:

$$AC_o = C((\mathbf{U}\ 19) * 3 - 1)$$
$$= 3\ \textit{random accesses} + 53\ \textit{sequential accesses} = 1.07\ \textit{sec}$$

Let's consider the case where `relation_h` is a hashed file, and `relation_i` is an ISAM file, but both of them are temporal relations with the update count of 14 according to the characteristics of database/relations. Then on the average, there are 28 overflow records for each tuple, since each **replace** operation inserts two versions into a temporal relation. We also assume that the size of `relation_i` is 3712 blocks, which is 128 blocks multiplied by 29, that the size of the temporary relation is 19 blocks, and that there are 1024 tuples in the temporary relation. Now the file primitive expression corresponding to the algebraic expression AE-5 becomes:

```
  Read   (Heap, 3712)          +
( Read   (Heap, 19) * 2 - 1    +
  Write  (Heap, 19) * 3 - 1 )  +
  Read   (Heap, 19)            +
  Read   (Hash, 28) * 1024
```

As in the previous example, the first **Read** primitive accounts for the **When** and the **Project** operations, the second **Read** and the **Write** primitives account for the **Temporary** operation, and the third and the fourth **Read** primitives account for the **Join** and the **Project** operations. This is transformed to the access path expression for input:

```
(U 3712)                          +
(U 19) * 2 - 1                    +
(U 19)                            +
(H 1 (P 28 (S 1) (P 1))) * 1024
```

and the access path expression for output:

```
(U 19) * 3 - 1
```

Then, the access cost for input is:

$$AC_i = C((\mathbf{U}\ 3712)) + C((\mathbf{U}\ 19) * 2 - 1) + C((\mathbf{U}\ 19))$$
$$+ C((\mathbf{H}\ 1\ (\mathbf{P}\ 28\ (\mathbf{S}\ 1)\ (\mathbf{P}\ 1)))) * 1024$$
$$= 29700\ \textit{random accesses} + 3764\ \textit{sequential accesses} = 999\ \textit{sec}$$

and the access cost for output is:

$$AC_o = C((\mathbf{U}\ 19) * 3 - 1)$$
$$= 3\ \textit{random accesses} + 53\ \textit{sequential accesses} = 1.07\ \textit{sec}$$

for output according to the characteristics of storage devices.

4.2. Validation

A prototype temporal DBMS was built by extending the snapshot DBMS INGRES. It supports the temporal query language TQuel, handling all four types of databases: snapshot, rollback, historical and temporal.

```
        range of h  is temporal_h          /* hashed on id */
        range of i  is temporal_i          /* ISAM   on id */

Q01 :   retrieve (h.id, h.seq)  where h.id = 500

Q02 :   retrieve (i.id, i.seq)  where i.id = 500

Q03 :   retrieve (h.id, h.seq)  as of "08:00 1/1/80"

Q04 :   retrieve (i.id, i.seq)  as of "08:00 1/1/80"

Q05 :   retrieve (h.id, h.seq)  where h.id = 500
              when  h overlap "now"

Q06 :   retrieve (i.id, i.seq)  where i.id = 500
              when  i overlap "now"

Q07 :   retrieve (h.id, h.seq)  where h.amount = 69400
              when  h overlap "now"

Q08 :   retrieve (i.id, i.seq)  where i.amount = 73700
              when  i overlap "now"

Q09 :   retrieve (h.id, i.id, i.amount)     where h.id = i.amount
              when  h overlap i and   i overlap "now"

Q10 :   retrieve (i.id, h.id, h.amount)     where i.id = h.amount
              when  h overlap i and   h overlap "now"

Q11 :   retrieve (h.id, h.seq, i.id, i.seq, i.amount)
              valid from begin of h   to end of i
              when  begin of h precede i
              as of "4:00 1/1/80"

Q12 :   retrieve (h.id, h.seq, i.id, i.seq, i.amount)
              valid from begin of (h overlap i)   to end of (h extend i)
              where h.id = 500  and   i.amount = 73700
              when  h overlap i
              as of "now"

Q13 :   retrieve (h.id, h.seq)  where h.id = 455
              when  "1/1/82" precede  end of h

Q14 :   retrieve (h.id, h.seq)  where h.amount = 10300
              when  "1/1/82" precede  end of h

Q15 :   retrieve (h.id, h.seq)  where h.amount = 10300
              as of "1/1/83"

Q16 :   retrieve (h.id, h.seq)  where h.amount = 10300
              when  "1/1/82" precede  end of h
              as of "1/1/83"
```

Figure 7: Benchmark Queries

We augment each tuple of a rollback or a temporal relation with two temporal attributes for *transaction time*, and each tuple of a historical or a temporal relation with one or two temporal attributes for *valid time*. The prototype also supports **append**, **delete**, and **replace** statements of TQuel for all four temporal types [Ahn & Snodgrass 1986].

To compare performance on different types of databases, we created test databases of all four temporal types: `Snapshot`, `Rollback`, `Historical`, and `Temporal`. For each of the four types, we created two databases, one with a 100% loading factor and the other with a 50% loading factor. A loading factor of 50%, for example, specifies that 50% of each primary data page is initially reserved free for future growth. Each database contains two relations, one with the structure of hashing and the other with the structure of ISAM.

A benchmark of sixteen queries, shown in Figure 7, was run to study the performance of the prototype on the test databases. The input costs for each type of databases with the update count of 0 and 14 are shown in Figure 8.

Type	Snapshot		Rollback				Historical				Temporal			
Loading	100 %	50 %	100 %		50 %		100 %		50 %		100 %		50 %	
Query	UC 0	UC 0	UC 0	UC 14	UC 0	UC 14	UC 0	UC 14	UC 0	UC 14	UC 0	UC 14	UC 0	UC 14
Q01	2	1	1	15	1	8	1	15	1	8	1	29	1	15
Q02	2	3	2	16	3	10	2	16	3	10	2	30	3	17
Q03	–	–	129	1927	257	2048	–	–	–	–	129	3717	257	3839
Q04	–	–	128	1920	256	2048	–	–	–	–	128	3712	256	3840
Q05	2	1	1	15	1	8	1	15	1	8	1	29	1	15
Q06	2	3	2	16	3	10	2	16	3	10	2	30	3	17
Q07	166	257	129	1927	257	2048	129	1927	257	2048	129	3717	257	3839
Q08	114	256	128	1920	256	2048	128	1920	256	2048	128	3712	256	3840
Q09	1585	1276	1141	17242	1271	10240	1197	17298	1327	10296	1200	33350	1333	19256
Q10	2214	3329	2177	18311	3329	12288	2233	18367	3385	12344	2233	34493	3385	21303
Q11	–	–	–	–	–	–	–	–	–	–	385	11141	769	11519
Q12	–	–	–	–	–	–	–	–	–	–	131	3743	259	3857
Q13	–	–	–	–	–	–	1	15	1	8	1	29	1	15
Q14	–	–	–	–	–	–	129	1927	257	2048	129	3717	257	3839
Q15	–	–	129	1927	257	2048	–	–	–	–	129	3717	257	3839
Q16	–	–	–	–	–	–	–	–	–	–	129	3717	257	3839

Notes :
'UC' denotes *Update Count.* '–' denotes *not applicable.*

Figure 8: Measured Input Costs for Four Types of Databases

The sample queries were also analyzed using the model discussed in Section 3. To compare the cost estimated from the analysis with the actual cost measured from the benchmark (Figure 8), we calculate the *error rate* as:

$$Error\ Rate = \frac{a - b}{b} \times 100\ \%$$

where

a = cost estimated from the analysis
b = cost measured from the benchmark

Figure 9 shows the error rate for each data point. It shows that error rates are generally within about 1% for the rollback, historical, and temporal databases. Interestingly, the biggest errors are found for the snapshot database. The reason is that a snapshot relation with 100% loading can hold 9 tuples, compared with 8 tuples for

other types of relations with time stamps in each tuple, but the larger number of tuples per block caused extra key collisions due to imperfect nature of the hash function used for hashing. For example, a snapshot relation which can hold 9 tuples per block consumed 166 blocks for 1024 tuples, not 114 tuples as expected for a perfect hashing [Sprugnoli 1977]. As a result, query Q07 costs 166 blocks accesses to scan a hashed relation, and query Q01 costs two block accesses, not one as expected for hashing, to retrieve a tuple through a hashed key. The unpredictability of key collisions is less visible for other types of relations, which hold a smaller number of tuples per block to incorporate time attributes, but it still contributes to discrepancies between the analysis results and the actual measurements.

Type	Snapshot		Rollback				Historical				Temporal			
Loading	100 %	50 %	100 %		50 %		100 %		50 %		100 %		50 %	
Query	U.C. 0	U.C. 0	U.C. 0	14	U.C. 0	14	U.C. 0	14	U.C. 0	14	U.C. 0	14	U.C. 0	14
Q01	-50	Z	Z	Z	Z	Z	Z	Z	Z	Z	Z	Z	Z	Z
Q02	Z	Z	Z	Z	Z	Z	Z	Z	Z	Z	Z	Z	Z	Z
Q03	–	–	-1	≈0	≈0	Z	–	–	–	–	-1	≈0	≈0	≈0
Q04	–	–	Z	Z	Z	Z	–	–	–	–	Z	Z	Z	Z
Q05	-50	Z	Z	Z	Z	Z	Z	Z	Z	Z	Z	Z	Z	Z
Q06	Z	Z	Z	Z	Z	Z	Z	Z	Z	Z	Z	Z	Z	Z
Q07	-31	-11	-1	≈0	≈0	Z	-1	≈0	≈0	Z	-1	≈0	≈0	≈0
Q08	Z	-11	Z	Z	Z	Z	Z	Z	Z	Z	Z	Z	Z	Z
Q09	-25	+3	+1	≈0	+1	Z	+1	≈0	+1	Z	+1	≈0	≈0	Z
Q10	≈0	+1	≈0	≈0	≈0	Z	≈0	≈0	≈0	Z	≈0	≈0	≈0	≈0
Q11	–	–	–	–	–	–	–	–	–	–	≈0	≈0	≈0	≈0
Q12	–	–	–	–	–	–	–	–	–	–	Z	Z	Z	Z
Q13	–	–	–	–	–	–	Z	Z	Z	Z	Z	Z	Z	Z
Q14	–	–	–	–	–	–	-1	≈0	≈0	Z	-1	≈0	≈0	≈0
Q15	–	–	-1	≈0	≈0	Z	–	–	–	–	-1	≈0	≈0	≈0
Q16	–	–	–	–	–	–	–	–	–	–	-1	≈0	≈0	≈0

Notes :

'U.C.' denotes *Update Count.* '–' denotes *not applicable.*

'Z' denotes *zero* error. '≈0' denotes *close to zero.*

Figure 9: Percentage Error Rates in the Analysis Results

We also measured the elapsed time to process the sample queries on the prototype. Figure 10 compares the measured time with the estimated time based on the characteristics of storage devices. This table shows that the differences between the measurement and the estimation is generally between 10 and 30%. There are many factors to affect the elapsed time to process a query, other than input and output costs; examples are the CPU speed, machine load, scheduling policy, and buffer management algorithms. Though we analyzed only input and output costs, we could still estimate the elapsed time rather closely.

Query	Update Count = 0			Update Count = 14		
	Measured (*sec*)	Estimated (*sec*)	Error Rate (%)	Measured (*sec*)	Estimated (*sec*)	Error Rate (%)
Q09	44.8	36.5	-18.5	1277	1001	-21.6
Q10	61.2	68.5	11.9	1187	1031	-13.1
Q11	7.8	7.1	-9	140	205	46.4
Q12	4.0	2.5	-37.5	62	69.2	11.6

Figure 10: Elapsed Time

5. Performance Analyzer

Based on our model consisting of a series of transformations, it is possible to construct a system that can automate the process of computing the I/O cost given a collection of TQuel queries as input. The internal structure of the performance analyzer is shown in Figure 11.

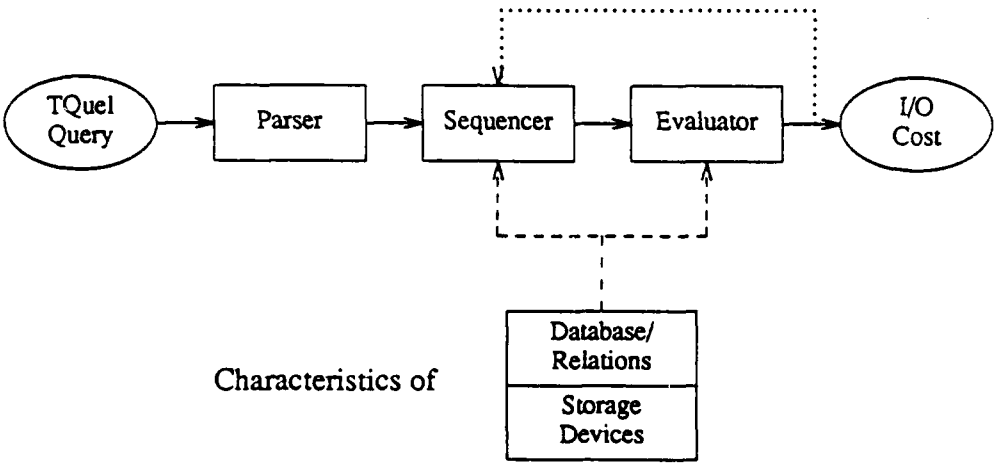


Figure 11: Performance Analyzer for TQuel Queries

The *parser* will take TQuel queries and generate a parse tree. The *sequencer* converts the tree into an algebraic expression consisting of algebraic operators and connectives as described in Section 3.1.1. Since TQuel is a non-procedural language based on the tuple calculus, there are many ways to process a TQuel query, and many variations of algebraic expressions. The sequencer is the embodiment of the query processing and optimization strategy for a particular database management system.

The resulting algebraic expression is processed by the *evaluator* to compute the input and output cost. The evaluator converts the algebraic expression to the file primitive expression based on the characteristics of database/relations. Next, it converts the file primitive expression to the access path expression, and eventually to the access cost based on the characteristics of storage devices.

The performance analyzer can be used to test and analyze various alternatives in the design of new access methods, database configurations, or query processing strategies, eliminating the tedious process of case by case implementation or simulation. In this paper, we analyzed the performance of sample queries manually, but in the same manner the analyzer would have employed.

The analyzer can be extended to be an optimization tool by providing a feedback path, as shown by a dotted line in Figure 11, from the evaluator output to the sequencer. The sequencer can generate all possible algebraic

expressions (or a significant subset of them) for an input parse tree, and can choose the one with the lowest input and output cost as computed by the evaluator. The algebraic expression chosen that way represents the best strategy to minimize the cost of processing the query.

6. Conclusions

Performance analysis of a database management system depends on the quality of the models to characterize various phases of query processing. We presented a model to analyze the input and output cost of temporal queries on the database with various access methods. The model consists of four transformations through a series of intermediate expressions based on the characteristics of database/relations and storage devices. A temporal query is mapped to an *algebraic expression* which is transformed to a *file primitive expression*. A file primitive expression, in turn, is transformed to an *access path expression*, and finally to the *access cost*. Since conventional databases are subsets of temporal databases, the model can be used to analyze the performance of conventional databases as well.

We showed examples of using the model to analyze the I/O cost of temporal queries. We validated the model by comparing the I/O cost estimated from the analysis with the actual cost measured from a prototype temporal DBMS. The result indicated that the cost of a query in terms of block access counts can be estimated quite accurately (generally within about 1%) using the model. Elapsed time to process a query, according to characteristics of storage devices, was generally between 10 and 30% of the actual measurements.

Our model, however, does not represent the query processing strategy of a DBMS, nor to estimate the CPU cost. While we believe that the benchmark in Section 4.2 covers representative operations in the temporal database, we have not included modification statements, aggregates, or queries involving more than two relations. We also limited our discussion to access methods of heap, hashing, and ISAM in this paper, though we have applied the model to the prototype that has been extended with a form of *two level storage structure* [Ahn 1986B]. In the future, we plan to analyze the performance of modification statements, queries with aggregates, and queries of more than two relations. We also plan to study the applicability of the model to temporal algebra based on *attribute versioning* [McKenzie 1988], as opposed to *tuple versioning*, and storage structures for optical disks. Finally, we are implementing the performance analyzer so that it may aid us in these experiments.

7. Bibliography

[Ahn 1986A] Ahn, I. *Towards an Implementation of Database Management Systems with Temporal Support*, in *Second International Conference on Data Engineering*, IEEE. Feb. 1986, pp. 374-381.

[Ahn 1986B] Ahn, I. *Performance Modeling and Access Methods for Temporal Database Management Systems*. PhD. Diss. Computer Science Department, University of North Carolina at Chapel Hill, Aug. 1986.

[Ahn & Snodgrass 1986] Ahn, I. and R. Snodgrass. *Performance Evaluation of a Temporal Database Management System*, in *Proceedings of ACM SIGMOD International Conference on Management of Data*, Washington, DC: May 1986, pp. 96-107.

[Ariav & Morgan 1982] Ariav, G. and H.L. Morgan. *MDM: Embedding the Time Dimension in Information Systems*. TR 82-03-01. Department of Decision Sciences, The Wharton School, University of Pennsylvania. 1982.

[Batory & Gotlieb 1982] Batory, D. and C. Gotlieb. *A Unifying Model of Physical Databases. ACM Transactions on Database Systems*, 7, No. 4, Dec. 1982, pp. 509-539.

[Batory 1985] Batory, D. *Modeling The Storage Architecture Of Commercial Database Systems. ACM Transactions on Database Systems*, 10, No. 4, Dec. 1985, pp. 463-528.

[Bolour et al. 1982] Bolour, A., T.L. Anderson, L.J. Dekeyser and H.K.T. Wong. *The Role of Time in Information Processing: A Survey. SigArt Newsletter*, 80, Apr. 1982, pp. 28-48.

[Ceri & Pelagatti 1984] Ceri, S. and G. Pelagatti. *Distributed Databases Principles & Systems*. NY: McGraw-Hill, 1984.

[Clifford & Tansel 1985] Clifford, J. and A. Tansel. *On An Algebra For Historical Relational Databases: Two Views*, in *Proceedings of ACM SIGMOD International Conference on Management of Data*, 1985, pp. 247-265.

[Gremillion 1982] Gremillion, L. *Designing a Bloom Filter for Differential File Access. Communications of the Association of Computing Machinery*, 25, No. 9, Sep. 1982, pp. 600-604.

[Held et al. 1975] Held, G.D., M. Stonebraker and E. Wong. *INGRES--A Relational Data Base Management System. Proceedings of the AFIPS 1975 National Computer Conference*, 44, May 1975, pp. 409-416.

[Hsiao & Harary 1970] Hsiao, D. and F. Harary. *A Formal System for Information Retrieval from Files. Communications of the Association of Computing Machinery*, 13, No. 2, Feb. 1970, pp. 67-73.

[March & Severance 1977] March, D. and D. Severance. *The Determination of Efficient Record Segmentations and Blocking Factors for Shared Files. ACM Transactions on Database Systems*, 2, No. 3, Sep. 1977, pp. 279-296.

[McKenzie 1986] McKenzie, E. *Bibliography: Temporal Databases. ACM SIGMOD Record*, 15, No. 4, Dec. 1986, pp. 40-52.

[McKenzie 1988] McKenzie, E. *An Incremental Temporal Relational Algebraic Language (in progress)*. PhD. Diss. Computer Science Department, University of North Carolina at Chapel Hill, 1988.

[Navathe & Ahmed 1986] Navathe, S.B. and R. Ahmed. *A Temporal Relational Model and a Query Language*. UF-CIS Technical Report TR-85-16. Computer and Information Sciences Department, University of Florida. Apr. 1986.

[Severance 1975] Severance, D. *A Parametric Model of Alternative File Structures. Information Systems*, 1, No. 2 (1975), pp. 51-55.

[Severance 1976] Severance, D. *Differential Files: Their Application to the Maintenance of Large Databases. ACM Transactions on Database Systems*, 1, No. 3, Sep. 1976, pp. 256-267.

[Snodgrass & Ahn 1985] Snodgrass, R. and I. Ahn. *A Taxonomy of Time in Databases*, in *Proceedings of ACM SIGMOD International Conference on Management of Data*, Ed. S. Navathe. Association for Computing Machinery. Austin, TX: May 1985, pp. 236-246.

[Snodgrass & Ahn 1986] Snodgrass, R. and I. Ahn. *Temporal Databases. IEEE Computer*, 19, No. 9, Sep. 1986, pp. 35-42.

[Snodgrass 1987] Snodgrass, R. *The Temporal Query Language TQuel. ACM Transactions on Database Systems*, 12, No. 2, June 1987, pp. 243-298.

[Sprugnoli 1977] Sprugnoli, R. *Perfect hash functions: A single probe retrieving method for static sets. Communications of the Association of Computing Machinery*, 20, No. 11, Nov. 1977, pp. 841-850.

[Stonebraker 1981] Stonebraker, M. *Operating System Support for Database Management. Communications of the Association of Computing Machinery*, 24, No. 7, July 1981, pp. 412-418.

[Wiederhold 1981] Wiederhold, G. *Databases for Health Care*. New York, NY: Springer-Verlag, 1981.

[Wong & Youssefi 1976] Wong, E. and K. Youssefi. *Decomposition - A Strategy for Query Processing. ACM Transactions on Database Systems*, 1, No. 3, Sep. 1976, pp. 223-240.

[Yao & Merten 1975] Yao, S. and A. Merten. *Selection Of File Organization Using An Analytic Model*, in *Proceedings of the Conference on Very Large Databases*, Sep. 1975, pp. 255-267.

[Yao 1977] Yao, S. *An Attribute Based Model for Database Access Cost Analysis. ACM Transactions on Database Systems*, 2, No. 1, Mar. 1977, pp. 45-67.

[Yao 1979] Yao, S. *Optimization of Query Evaluation Algorithms. ACM Transactions on Database Systems*, 4, No. 2, June 1979, pp. 133-155.

Appendix

A.1. BNF Syntax of the Algebraic Expression

```
<alg exp>        ::=   <query>
                   |   <alg exp> <query>

<query>          ::=   { <access> }

<access>         ::=   <access term> | <access> <access term>
<access term>    ::=   <term> | [ <term> ]

<term>           ::=   <label oper> | <term> <order> <label oper>

<label oper>     ::=   <oper> | <label> : <oper>
<order>          ::=   ; | ,
<label>          ::=   <identifier>

<oper>           ::=   <Conventional> | <Temporal> | <Auxiliary>

<Conventional>   ::=   Select      ( <rel> , <predicate> )
                   |   Project     ( <rel> , <attribute list> )
                   |   Join        ( <rel> , <rel> , <join method> , <predicate> )
                   |   Union       ( <rel> , <rel> )
                   |   Difference  ( <rel> , <rel> )

<Temporal>       ::=   When    ( <rel> , <temporal pred> )
                   |   AsOf    ( <rel> , <event expr> , <event expr> )
                   |   Valid   ( <rel> , <FTA> , <event expr> )

<Auxiliary>      ::=   Temporary  ( <label> )
                   |   Sort       ( <rel> , <attribute list> )
                   |   Reformat   ( <rel> , <storage spec> , <attribute list> )
<FTA>            ::=   From | To | At

<attribute list> ::=   <attribute>
                   |   <attribute list> , <attribute>
<rel>            ::=   <rel identifier> | <label>
```

In this description, <temporal pred> is a *temporal predicate* involving time attributes and temporal predicate operators such as **precede** and **overlap** in TQuel. <event expr> is an *event expression* involving time attributes and temporal constructor operators such as **extend** and **overlap** in TQuel, which yields a time value as its result. <storage spec> specifies one of the storage structures such as `heap`, `hash`, `isam`, or a `btree`.

A.2. BNF Syntax of the File Primitive Expression

```
<fpe>                 ::=   <term>
                      |     <fpe>  <additive op>  <term>
<term>                ::=   <primitive>
                      |     <term>  <multiplicative op>  <primitive>

<primitive>           ::=   <operator>  (  <access method>  <parameter list>  )
                      |     (  <fpe>  )

<operator>            ::=   Read
                      |     Write

<parameter list>      ::=   <parameter>
                      |     <parameter list>  <parameter>

<parameter>           ::=   <integer constant>

<additive op>         ::=   +       |       -

<multiplicative op>   ::=   *       |       /

<access method>       ::=   heap | hash | isam | btree
```

A.3. BNF Syntax to Represent the Characteristics of Database/Relations

```
<database>       ::=  <name> , <relation list>
<name>           ::=  <identifier>

<relation list>  ::=  <relation> | <relation list> , <relation>

<relation>       ::=  <name>          , <attribute list> ,
                      <temporal type> , <storage type> ,
                      <tuple count>    , <update count> ,
                      <loading factor> , <block Size> ,
                      <key list>

<attribute list> ::=  <attribute> | <attribute list> , <attribute>

<attribute>      ::=  <name>          , <value type> ,
                      <length>        , <selectivity> ,
                      <volatility>

<temporal type>  ::=  snapshot                | rollback          |
                      historical interval     | historical event  |
                      temporal interval       | temporal event

<storage type>   ::=  heap | hash | isam | btree

<key list>       ::=  <key> | <key list> , <key>
<key>            ::=  ( <name> , <attribute list> )

<value type>     ::=  type integer      | type rational    |
                      type string       | type boolean     |
                      type time

<tuple count>    ::=  <integer constant>
<update count>   ::=  <integer constant>
<loading factor> ::=  <float constant>
<block Size>     ::=  <integer constant>
<length>         ::=  <integer constant>
<selectivity>    ::=  <float constant>
<volatility>     ::=  <float constant>
```

In this description, a database consists of a name and a set of relations. Each relation consists of a name and various information on the relation. For example, <temporal type> specifies one of six possible temporal types, and <storage type> specifies the storage structure of the relation.

A.4. BNF Syntax of the File Path Expression for a Single File

```
<APE>          ::=   <file path>

<file path>    ::=   ( <descriptor> <chains> )
<descriptor>   ::=   <Mode> <count>

<chains>       ::=   <chain>
                 |   <chains> <chain>
<chain>        ::=   ( <descriptor> <nodes> )

<nodes>        ::=   <node>
                 |   <nodes> <node>
<node>         ::=   ( <descriptor> )

<Mode>         ::=   H
                 |   P
                 |   A
                 |   S
                 |   M
                 |   O
                 |   U
<count>        ::=   <integer constant>
```

A.5. BNF Syntax of the Access Path Expression for Multiple Files

```
<APE>                ::=   <term>
                       |   <APE> <additive op> <term>

<term>               ::=   <access path>
                       |   <term> <multiplicative op> <access path>

<additive op>        ::=   +     |     -
<multiplicative op>  ::=   *     |     /

<access path>        ::=   <file path>
                       |   [ <access path> <single> <order> <file path> ]
                       |   ( <APE> )

<single>             ::=   ?
<order>              ::=   ;     |     ,
```

END
DATE
FILMD
3-88
DTIC